Coding Standards Document

CHAPTER

6

IN THIS CHAPTER

Introduction

This document describes the coding standards for Delphi programming as used in *Delphi 5 Developer's Guide*. In general, this document follows the often "unspoken" formatting guidelines used by Borland International with a few minor exceptions. The purpose for including this document in *Delphi 5 Developer's Guide* is to present a method by which development teams can enforce a consistent style to the coding they do. The intent is to make it so that every programmer on a team can understand the code being written by other programmers. This is accomplished by making the code more readable by use of consistency.

This document by no means includes everything that might exist in a coding standard. However, it does contain enough detail to get you started. Feel free to use and modify these standards to fit your needs. We don't recommend, however, that you deviate too far from the standards used by Borland's development staff. We recommend this because as you bring new programmers to your team, the standards that they're most likely to be most familiar with are Borland's. Like most coding standards documents, this document will evolve as needed. Therefore, you'll find the most updated version online at `www.xapware.com/ddg`.

This document does not cover *user interface standards*. This is a separate but equally important topic. Enough third-party books and Microsoft documentation cover such guidelines that we decided not to replicate this information but rather refer you to the Microsoft Developers Network and other sources where that information is available.

General Source Code Formatting Rules

Indentation

Indenting shall be two spaces per level. Do not save tab characters to source files. The reason for this is because tab characters are expanded to different widths with different users' settings and by different source management utilities (print, archive, version control, and so on).

You can disable saving tab characters by turning off the Use Tab Character and Optimal Fill check boxes on the General page of the Editor Properties dialog box (accessed via Tools, Editor Options).

Margins

Margins will be set to 80 characters. In general, source shall not exceed this margin, with the exception to finish a word. However, this guideline is somewhat flexible. Wherever possible, statements that extend beyond one line shall be wrapped after a comma or an operator. When a statement is wrapped, it shall be indented two characters from the original statement line.

begin..end Pair

The `begin` statement appears on its own line. For example, the following first line is incorrect; the second line is correct:

```
for I := 0 to 10 do begin // Incorrect, begin on same line as for

for I := 0 to 10 do          // Correct, begin appears on a separate line
begin
```

An exception to this rule is when the `begin` statement appears as part of an `else` clause. Here's an example:

```
if some statement = then
begin
...
end
else begin
  SomeOtherStatement;
end;
```

The `end` statement always appears on its own line.

When the `begin` statement is not part of an `else` clause, the corresponding `end` statement is always indented to match its `begin` part.

Object Pascal

Parentheses

There shall never be white space between an open parenthesis and the next character. Likewise, there shall never be white space between a closed parenthesis and the previous character. The following example illustrates incorrect and correct spacing with regard to parentheses:

```
CallProc( AParameter );  // incorrect
CallProc(AParameter);    // correct
```

Never include extraneous parentheses in a statement. Parentheses shall only be used where required to achieve the intended meaning in source code. The following examples illustrate incorrect and correct usage:

```
if (I = 42) then         // incorrect - extraneous parentheses
if (I = 42) or (J = 42) then  // correct - parentheses required
```

Reserved Words and Key Words

Object Pascal language reserved words and key words shall always be completely lowercase.

Procedures and Functions (Routines)

Naming/Formatting

Routine names shall always begin with a capital letter and be camel-capped for readability. The following is an example of an incorrectly formatted procedure name:

```
procedure thisisapoorlyformattedroutinename;
```

This is an example of an appropriately capitalized routine name:

```
procedure ThisIsMuchMoreReadableRoutineName;
```

Routines shall be given names meaningful to their content. Routines that cause an action to occur will be prefixed with the action verb. Here's an example:

```
procedure FormatHardDrive;
```

Routines that set values of input parameters shall be prefixed with the word `set`:

```
procedure SetUserName;
```

Routines that retrieve a value shall be prefixed with the word `get`:

```
function GetUserName: string;
```

Formal Parameters

Formatting

Where possible, formal parameters of the same type shall be combined into one statement:

```
procedure Foo(Param1, Param2, Param3: Integer; Param4: string);
```

Naming

All formal parameter names shall be meaningful to their purpose and typically will be based off the name of the identifier that was passed to the routine. When appropriate, parameter names shall be prefixed with the character `A`:

```
procedure SomeProc(AUserName: string; AUserAge: integer);
```

The `A` prefix is a convention to disambiguate when the parameter name is the same as a property or field name in the class.

Ordering of Parameters

The following formal parameter ordering emphasizes taking advantage of register calling conventions calls.

Most frequently used (by the caller) parameters shall be in the first parameter slots. Less frequently used parameters shall be listed after that in left-to-right order.

Input lists shall exist before output lists in left-to-right order.

Place most generic parameters before most specific parameters in left-to-right order. For example: `SomeProc(APlanet, AContinent, ACountry, AState, ACity)`.

Exceptions to the ordering rule are possible, such as in the case of event handlers, where a parameter named `Sender` of type `TObject` is often passed as the first parameter.

Constant Parameters

When parameters of a record, array, `ShortString`, or interface type are unmodified by a routine, the formal parameters for that routine shall mark the parameter as `const`. This ensures that the compiler will generate code to pass these unmodified parameters in the most efficient manner.

Parameters of other types may optionally be marked as `const` if they're unmodified by a routine. Although this will have no effect on efficiency, it provides more information about parameter use to the caller of the routine.

Name Collisions

When using two units that each contain a routine of the same name, the routine residing in the unit appearing last in the `uses` clause will be invoked if you call that routine. To avoid these `uses` clause–dependent ambiguities, always prefix such method calls with the intended unit name. Here are two examples:

```
SysUtils.FindClose(SR);
```

and

```
Windows.FindClose(Handle);
```

Variables

Variable Naming and Formatting

Variables shall be given names meaningful to their purpose.

Loop control variables are generally given a single character name such as `I`, `J`, or `K`. It's acceptable to use a more meaningful name as well, such as `UserIndex`.

Boolean variable names must be descriptive enough so that the meanings of `True` and `False` values will be clear.

Local Variables

Local variables used within procedures follow the same usage and naming conventions for all other variables. Temporary variables shall be named appropriately.

When necessary, initialization of local variables will occur immediately upon entry into the routine. Local AnsiString variables are automatically initialized to an empty string, local interface and dispinterface type variables are automatically initialized to nil, and local Variant and OleVariant type variables are automatically initialized to Unassigned.

Use of Global Variables

Use of global variables is discouraged. However, they may be used when necessary. When this is the case, you're encouraged to keep global variables within the context in which they're used. For example, a global variable may be global only within the scope of a single unit's implementation section.

Global data that's intended to be used by a number of units shall be moved into a common unit used by all.

Global data may be initialized with a value directly in the var section. Bear in mind that all global data is automatically zero initialized; therefore, do not initialize global variables to "empty" values such as 0, nil, '', Unassigned, and so on. One reason for this is because zero-initialized global data occupies no space in the EXE file. Zero-initialized data is stored in a virtual data segment that's allocated only in memory when the application starts up. Nonzero initialized global data occupies space in the EXE file on disk.

Types

Capitalization Convention

Type names that are reserved words shall be completely lowercase. Win32 API types are generally completely uppercase, and you shall follow the convention for a particular type name shown in the Windows.pas or other API unit. For other variable names, the first letter shall be uppercase, and the rest shall be camel-capped for clarity. Here are some examples:

```
var
  MyString: string;     // reserved word
  WindowHandle: HWND;  // Win32 API type
  I: Integer;          // type identifier introduced in System unit
```

Floating-Point Types

Use of the Real type is discouraged because it existed only for backward compatibility with older Pascal code. Although it's now the same as Double, this fact may be confusing to other developers. Use Double for general-purpose floating-point needs. Also, Double is what the processor instructions and busses are optimized for and is an IEEE-defined standard data format. Use Extended only when more range is required than that offered by Double. Extended is an Intel-specified type and is not supported in Java. Use Single only when the physical byte size of the floating-point variable is significant (such as when using other-language DLLs).

Enumerated Types

Names for enumerated types must be meaningful to the purpose of the enumeration. The type name must be prefixed with the `T` character to annotate it as a type declaration. The identifier list of the enumerated type must contain a lowercase two-to-three-character prefix that relates it to the original enumerated type name. Here's an example:

```
TSongType = (stRock, stClassical, stCountry, stAlternative, stHeavyMetal,
     stRB);
```

Variable instances of an enumerated type will be given the same name as the type without the `T` prefix (`SongType`) unless there's a reason to give the variable a more specific name, such as `FavoriteSongType1`, `FavoriteSongType2`, and so on.

Variant and OleVariant Types

The use of the `Variant` and `OleVariant` types is discouraged in general, but these types are necessary for programming when data types are known only at runtime, as is often the case in COM and database development. Use `OleVariant` for COM-based programming such as Automation and ActiveX controls, and use `Variant` for non-COM programming. The reason is that a `Variant` can store native Delphi strings efficiently (like a `string var`), but `OleVariant` converts all strings to OLE strings (`WideChar` strings) and are not reference counted; instead, they're always copied.

Structured Types

Array Types

Names for array types must be meaningful to the purpose for the array. The type name must be prefixed with a `T` character. If a pointer to the array type is declared, it must be prefixed with the character `P` and declared immediately prior to the type declaration. Here's an example:

```
type
  PCycleArray = ^TCycleArray;
  TCycleArray = array[1..100] of integer;
```

When practical, variable instances of the array type shall be given the same name as the type name without the `T` prefix.

Record Types

A record type shall be given a name meaningful to its purpose. The type declaration must be prefixed with the character `T`. If a pointer to the record type is declared, it must be prefixed with the character `P` and declared immediately prior to the type declaration. The type declaration for each element may be optionally aligned in a column to the right. Here's an example:

```
type
```

```
PEmployee = ^TEmployee;

TEmployee = record
  EmployeeName: string
  EmployeeRate:  Double;
end;
```

Statements

if Statements

The most likely case to execute in an `if/then/else` statement shall be placed in the `then` clause, with less likely cases residing in the `else` clause(s).

Try to avoid chaining `if` statements and use `case` statements instead if at all possible.

Do not nest `if` statements more than five levels deep. Create a clearer approach to the code.

Do not use extraneous parentheses in an `if` statement.

If multiple conditions are being tested in an `if` statement, conditions shall be arranged from left to right in order of least to most computation intensive. This enables your code to take advantage of short-circuit Boolean evaluation logic built into the compiler. For example, if `Condition1` is faster than `Condition2`, and `Condition2` is faster than `Condition3`, then the `if` statement shall be constructed as follows:

```
if Condition1 and Condition2 and Condition3 then
```

case Statements

General Topics

The individual cases in a `case` statement shall be ordered by the case constant either numerically or alphabetically.

The actions statements of each case shall be kept simple and generally shall not exceed four to five lines of code. If the actions are more complex, the code shall be placed in a separate procedure or function.

The `else` clause of a `case` statement shall be used only for legitimate defaults or to detect errors.

Formatting

`case` statements follow the same formatting rules as other constructs in regards to indentation and naming conventions.

while Statements

The use of the `Exit` procedure to exit a `while` loop is discouraged; when possible, you shall exit the loop using only the loop condition.

All initialization code for a `while` loop shall occur directly before entering the `while` loop and shall not be separated by other nonrelated statements.

Any ending housekeeping shall be done immediately following the loop.

for Statements

`for` statements shall be used in place of `while` statements when the code must execute for a known number of increments.

repeat Statements

`repeat` statements are similar to `while` loops and shall follow the same general guidelines.

with Statements

General Topics

The `with` statement shall be used sparingly and with considerable caution. Avoid overuse of `with` statements and beware of using multiple objects, records, and so on in the `with` statement. For example,

```
with Record1, Record2 do
```

can confuse the programmer and can easily lead to difficult-to-detect bugs.

Formatting

`with` statements follow the same formatting rules in regard to naming conventions and indentation as described previously in this document.

Structured Exception Handling

General Topics

Exception handling shall be used abundantly for both error correction and resource protection. This means that in all cases where resources are allocated, a `try..finally` must be used to ensure proper deallocation of the resource. The exception to this involves cases where resources are allocated/freed in the initialization/finalization of a unit or the constructor/destructor of an object.

Use of try..finally

Where possible, each allocation shall be matched with a `try..finally` construct. For example, the following code could lead to possible bugs:

```
SomeClass1 := TSomeClass.Create;
SomeClass2 := TSomeClass.Create;
try
   { do some code }
finally
```

```
  SomeClass1.Free;
  SomeClass2.Free;
end;
```

A safer approach to the preceding allocation would be this:

```
SomeClass1 := TSomeClass.Create
try
  SomeClass2 := TSomeClass.Create;
  try
     { do some code }
  finally
    SomeClass2.Free;
  end;
finally
  SomeClass1.Free;
end;
```

Use of try..except

Use `try..except` only when you want to perform some task when an exception is raised. In general, you shall not use `try..except` to simply show an error message on the screen because that will be done automatically in the context of an application by the `Application` object. If you want to invoke the default exception handling after you've performed some task in the `except` clause, use `raise` to reraise the exception to the next handler.

Use of try..except..else

The use of the `else` clause with `try..except` is discouraged because it will block all exceptions, even those for which you may not be prepared.

Classes

Naming/Formatting

Type names for classes shall be meaningful to the purpose of the class. The type name must have the `T` prefix to annotate it as a type definition. Here's an example:

```
type
  TCustomer = class(TObject)
```

Instance names for classes will generally match the type name of the class without the `T` prefix:

```
var
  Customer: TCustomer;
```

NOTE

See the section “Component Type Naming Standards” for further information on naming components.

Fields

Naming/Formatting

Class field names follow the same naming conventions as variable identifiers, except they’re prefixed with the `F` annotation to signify that they’re field names.

Visibility

All fields shall be private. Fields that are accessible outside the class scope shall be made accessible through the use of a property.

Methods

Naming/Formatting

Method names follow the same naming conventions as described for procedures and functions in this document.

Use of Static Methods

Use static methods when you do not intend for a method to be overridden by descendant classes.

Use of Virtual/Dynamic Methods

Use virtual methods when you intend for a method to be overridden by descendant classes. Dynamic methods shall only be used on classes of which there will be many descendants (direct or indirect). For example, when working with a class that contains one infrequently overridden method and 100 descendent classes, you shall make the method dynamic to reduce the memory use by the 100 descendent classes.

Use of Abstract Methods

Do not use abstract methods on classes of which instances will be created. Use abstract methods only on base classes that will never be created.

Property-Access Methods

All access methods must appear in the `private` or `protected` sections of the class definition.

The naming conventions for property-access methods follow the same rules as for procedures and functions. The read accessor method (reader method) must be prefixed with the word `Get`.

The write accessor method (writer method) must be prefixed with the word `Set`. The parameter for the writer method shall have the name `Value`, and its type shall be that of the property it represents. Here's an example:

```
TSomeClass = class(TObject)
private
  FSomeField: Integer;
protected
  function GetSomeField: Integer;
  procedure SetSomeField( Value: Integer);
public
  property SomeField: Integer read GetSomeField write SetSomeField;
end;
```

Properties

Naming/Formatting

Properties that serve as accessors to private fields will be named the same as the fields they represent, without the `F` annotator.

Property names shall be nouns, not verbs. Properties represent data; methods represent actions.

Array property names shall be plural. Normal property names shall be singular.

Use of Access Methods

Although not required, it's encouraged that you use, at a minimum, a write access method for properties that represent a private field.

Files

Project Files

Naming

Project files shall be given descriptive names. For example, *The Delphi 5 Developer's Guide Bug Manager* is given the project name `DDGBugs.dpr`. A system information program shall be given a name such as `SysInfo.dpr`.

Form Files

Naming

A form file shall be given a name descriptive of the form's purpose, postfixed with the characters `Frm`. For example, an About form would have the filename `AboutFrm.dpr`, and a Main form would have the filename `MainFrm.dpr`.

Data Module Files

Naming

A data module shall be given a name that's descriptive of the data module's purpose. The name shall be postfixed with the characters `DM`. For example, a Customers data module will have the form filename `CustomersDM.dfm`.

Remote Data Module Files

Naming

A remote data module shall be given a name that's descriptive of the remote data module's purpose. The name shall be postfixed with the characters `RDM`. For example, a Customers remote data module would have the form filename `CustomersRDM.dfm`.

Unit Files

General Unit Structure

Unit Name

Unit files shall be given descriptive names. For example, the unit containing an application's main form might be called `MainFrm.pas`.

The uses Clause

A `uses` clause in the `interface` section shall only contain units required by code in the `interface` section. Remove any extraneous unit names that might have been automatically inserted by Delphi.

A `uses` clause in the `implementation` section shall only contain units required by code in the `implementation` section. Remove any extraneous unit names.

The interface Section

The `interface` section shall contain declarations for only those types, variables, procedure/function forward declarations, and so on that are to be accessible by external units. Otherwise, these declarations shall go into the `implementation` section.

The implementation Section

The `implementation` section shall contain any declarations for types, variables, procedures/functions, and so on that are private to the containing unit.

The initialization Section

Do not place time-intensive code in the `initialization` section of a unit. This will cause the application to seem sluggish upon startup.

The finalization Section

Make sure you deallocate any items you allocated in the `initialization` section.

Form Units

Naming

A unit file for a form shall be given the same name as its corresponding form file. For example, an About form would have the unit name `AboutFrm.pas`, and a Main form would have the unit filename `MainFrm.pas`.

Data Module Units

Naming

Unit files or data modules shall be given the same names as their corresponding form files. For example, a Customers data module unit would have the unit name `CustomersDM.pas`.

General-purpose Units

Naming

A general-purpose unit shall be given a name meaningful to the unit's purpose. For example, a utilities unit would be given the name `BugUtilities.pas`, and a unit containing global variables would be given the name `CustomerGlobals.pas`.

Keep in mind that unit names must be unique across all packages used by a project. Generic or common unit names are not recommended.

Component Units

Naming

Component units shall be placed in a separate directory to distinguish them as units defining components or sets of components. They shall never be placed in the same directory as the project. The unit name must be meaningful to its content.

NOTE

See the section "User-Defined Components" for further information on component-naming standards.

File Headers

Use of informational file headers is encouraged for all source files, project files, units, and so on. A proper file header must contain the following information:

```
{
Copyright © YEAR by AUTHORS
}
```

Forms and Data Modules

Forms

Form Type Naming Standard

Form types shall be given names descriptive of the form's purpose. The type definition shall be prefixed with a `T`, and a descriptive name shall follow the prefix. Finally, `Form` shall postfix the descriptive name. For example, the type name for an About form would be

```
TAboutForm = class(TForm)
```

A main form definition would be

```
TMainForm = class(TForm)
```

The customer entry form would have a name such as

```
TCustomerEntryForm = class(TForm)
```

Form Instance Naming Standard

Form instances shall be named the same as their corresponding types, without the `T` prefix. For example, for the preceding form types, the instance names are as follows:

Type Name	*Instance Name*
`TAboutForm`	`AboutForm`
`TMainForm`	`MainForm`
`TCustomerEntryForm`	`CustomerEntryForm`

Auto-creating Forms

Only the main form shall be autocreated unless there's a good reason to do otherwise. All other forms must be removed from the Autocreate list in the Project Options dialog box. See the following section for more information.

Modal Form Instantiation Functions

All form units shall contain a form-instantiation function that creates, sets up, and shows the form modally as well as frees the form. This function shall return the modal result returned by the form. Parameters passed to this function shall follow the parameter-passing standard specified in this document. This functionality is to be encapsulated in this way to facilitate code reuse and maintenance.

The form variable shall be removed from the unit and declared locally in the form-instantiation function. (Note that this requires that the form be removed from the Autocreate list in the Project Options dialog box. See "Autocreating Forms" earlier in this document.)

For example, the following unit illustrates such a function for a GetUserData form:

```
unit UserDataFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TUserDataForm = class(TForm)
    edtUserName: TEdit;
    edtUserID: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

function GetUserData(var aUserName: String; var aUserID: Integer): Word;

implementation
{$R *.DFM}

function GetUserData(var aUserName: String; var aUserID: Integer): Word;
var
  UserDataForm: TUserDataForm;
begin
  UserDataForm := TUserDataForm.Create(Application);
  try
    UserDataForm.Caption := 'Getting User Data';
    Result := UserDataForm.ShowModal;
    if ( Result = mrOK ) then begin
      aUserName := UserDataForm.edtUserName.Text;
      aUserID   := StrToInt(UserDataForm.edtUserID.Text);
    end;
  finally
    UserDataForm.Free;
  end;
end;

end.
```

Data Modules

Data Module Naming Standard

A DataModule type shall be given a name descriptive of the data module's purpose. The type definition shall be prefixed with a T, and a descriptive name shall follow the prefix. Finally, the name shall be postfixed with the word DataModule. For example, the type name for a Customer data module would be something such as this:

```
TCustomerDataModule = class(TDataModule)
```

Similarly, an Orders data module might have the following name:

```
TOrdersDataModule = class(TDataModule)
```

Data Module Instance Naming Standard

Data module instances will be named the same as their corresponding types, without the T prefix. For example, for the preceding form types, the instance names are as follows:

Type Name	*Instance Name*
TCustomerDataModule	CustomerDataModule
TOrdersDataModule	OrdersDataModule

Packages

Use of Runtime Versus Design Packages

Runtime packages shall contain only units/components required by other components in that package. Other units containing property/component editors and other design-only code shall be placed into a design package. Registration units shall also be placed into a design package.

File Naming Standards

Packages shall be named according to the following templates:

- *iii*lib*vv*.dpk (design package)
- *iii*std*vv*.dpk (runtime package)

Here, the characters *iii* signify a three-character identifying prefix. This prefix may be used to identify a company, person, or any other identifying entity.

The characters *vv* signify a version for the package corresponding to the Delphi version for which the package is intended.

Note that the package name contains either lib or std to signify it as a runtime or design-time package.

In cases where there are both design-time and runtime packages, the files shall be named similarly. For example, packages for *Delphi 5 Developer's Guide* are named as follows:

- `DdgLib50.dpk` (design package)
- `DdgStd50.dpk` (runtime package)

Components

User-Defined Components

Component Type Naming Standards

Components shall be named similarly to classes as defined in the "Classes" section, with the exception that components are given a three-character identifying prefix. This prefix may be used to identify a company, person, or any other entity. For example, a clock component written for *Delphi 5 Developer's Guide* would be defined as follows:

```
TddgClock = class(TComponent)
```

Note that the three-character prefix is in lowercase.

Component Units

Component units shall contain only one major component. A *major component* is any component that appears on the Component Palette. Any ancillary components/objects may also reside in the same unit as the major component.

Use of Registration Units

The registration procedure for components shall be removed from the component unit and placed in a separate unit. This registration unit shall be used to register any components, property editors, component editors, experts, and so on.

Component registering shall be done only in the design packages; therefore, the registration unit shall be contained in the design package and not in the runtime package.

It's suggested that registration units be named as follows:

```
XxxReg.pas
```

Here, *`Xxx`* is a three-character prefix used to identify a company, person, or any other entity. For example, the registration unit for the components in the *Delphi 5 Developer's Guide* would be named `DdgReg.pas`.

Component Instance Naming Conventions

All componentsmust be given descriptive names. No components shall be left with the default names assigned to them by Delphi. Components shall be named using a variation of the

Hungarian naming convention. According to this standard, the component name shall consist of two parts: a component type prefix and qualifier name.

Component Type Prefixes

The component type prefix is a set of lower case letters that represent the component type. For example, the following are valid component type prefixes for the components specified.

TButton	btn
TEdit	edt
TSpeedButton	spdbtn
TListBox	lstbx

As shown above, the component type prefix is created by modifying the component type name (ie: TButton, TEdit) to a prefix. The following rules illustrate how to define a component type prefix:

1. Remove any "T" prefixes from the components type name. For example, "TButton" becomes "Button"
2. Remove any vowels from the name formed in step 1 with the exception of the first vowel. For example, "Button" becomes "bttn" and "edit" becomes "edt."
3. Suppress double consonants. For example, "bttn" becomes "btn."
4. If a naming conflict occurs, start adding vowels to the prefix for one of the components. For example, if a new component "TBatton" is added, it will conflict with "TButton." Therefore, the prefix for "TBatton" becomes "batn."

Component Qualifier Name

The component qualifier name shall be a descriptive of the component's purpose. For example, a TButton component with the purpose of closing a form would have the name "btnClose." A TEdit component used for editing the first name of a person would have the name "edtFirstName."

Coding Standards Document Updates

This document will be updated regularly to reflect changes and enhancements to the Object Pascal language and Visual Component Library. You can retrieve updates at `http://www.xapware.com/ddg`.